Epimenides, Gödel, Turing: an Eternal Gölden Twist [0]

Eric C.R. Hehner

Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

Abstract: The Halting Problem is a version of the Liar's Paradox. We examine specifications for dependence on the agent that performs them. We look at the consequences for the Church-Turing Thesis and for the Halting Problem.

Epimenides

An ancient Cretan named Epimenides is reported to have said "All Cretans are liars." [5]. This is supposed to be self-contradictory, but it misses the mark. If there is any other Cretan, and that Cretan is a truth-teller, then Epimenides' sentence is simply false: Epimenides is a liar, but not all Cretans are liars. Saint Paul missed the point completely, taking Epimenides' statement at face value, and elaborating: "It was one of themselves, one of their own prophets, who said, "Cretans were never anything but liars, dangerous animals, and lazy": and that is a true statement." [1]. I will refer to the simpler sentence

This sentence is false.

as the Liar's Paradox. If that sentence is true, then, according to the sentence, it is false. If it is false, then it is true. That simple sentence is self-contradictory.

I give the sentence a name, say L for Liar.

L: L is false.

As a mathematical formula, it becomes

$L = (L=false)$

As an equation in unknown L, it has no solution, because the equation is $false$ regardless of whether L is $true$ or $false$. As a definition or specification of L it is called "inconsistent". (I am using italic $true$ and $false$ for the binary values representing truth and falsity.)

A slightly more complicated version presents the inconsistency as two sentences.

The next sentence is true.

The previous sentence is false.

Naming the first sentence B and the second G, as mathematical formulas, they become

$B = (G=true)$

$G = (B=false)$

These two equations in the two unknowns B and G have no solution: there is no assignment of binary values to B and G that satisfies the two equations. They are inconsistent. If you look at either one of the sentences alone, there is no inconsistency. It may make sense to say that the next sentence is true, and it may make sense to say that the previous sentence is false. But together they are inconsistent.

Let me complicate this inconsistency by adding a parameter, so B can say whether any sentence is true, not just sentence G. To reduce contention over truth and falsity, I will stick with mathematical sentences, otherwise known as binary expressions (allowing subexpressions of any type, including functions). To pass sentences as data, we need to encode them in some way. The easiest encoding is as a text (character string). Now B becomes a function from texts to binary values, and the pair of sentences become

$B(t)$ = *true* if text t represents a binary expression with value *true* ;
false otherwise
G = " $B(G) =$ *false* "

I have made two definitions: B and G . Since G is just a text, there cannot be anything wrong with its definition; it represents the binary expression $B(G) =$ *false* . But the definition of B , no matter how carefully worded, no matter how clear it sounds, conceals an inconsistency. I am not concerned with computing B ; I just want to define a mathematical function. The parameter allows us to show a large number of examples, like B ("0=0") = *true* and B ("0=1") = *false* , which are not problematic. They may fool us into believing that the definition of B makes sense. But they are irrelevant. The inconsistency is revealed by applying B to G . If $B(G) =$ *true* , then G represents a *false* expression, so $B(G)$ should be *false* . If $B(G) =$ *false* , then G represents a *true* expression, so $B(G)$ should be *true* . The inconsistency is the same as in the unparameterized, unencoded version of the Liar's Paradox.

Gödel

The Liar's Paradox is about truth. Gödel used the same self-contradictory construction to talk about provability [6]. He used a numeric, rather than text, encoding of sentences, and he used the name *Bew* (short for "Beweisbar", which is German for "Provable") for a function similar to B . The sentence encoded by G is popularly called "the Gödel sentence". With our notations and encoding, B and G become

$B(t)$ = *true* if text t represents a provable binary expression;
false otherwise
G = " $B(G) =$ *false* "

What is the value of $B(G)$? If we suppose $B(G)$ is *true* , then G represents a *false* sentence, and in a consistent logic, no *false* sentence is provable, so $B(G)$ should be *false* . If we suppose $B(G)$ is *false* , then G represents a *true* sentence, and in a complete logic, all *true* sentences are provable, so $B(G)$ should be *true* . Gödel concluded that if a logic is expressive enough to define B , then the logic is either inconsistent or incomplete.

Most of Gödel's paper [6] is devoted to showing how to define *Bew* . His definition was equivalent to programming a prover, using a functional language, namely the logic of *Principia Mathematica* [14] (hence the name of his paper). That was an amazing piece of work. But Gödel needn't have gone to so much trouble. *Bew* is defined to apply to all sentence encodings. But there is only one sentence encoding he wants to apply it to. For a single sentence, we don't need a sentence encoding. Define

B = *true* if G is a provable binary expression;
false otherwise
G = (B=*false*)

What is the value of B ? If we suppose B is *true* , then G is a *false* sentence, and in a consistent logic, no *false* sentence is provable, so B should be *false* . If we suppose B is *false* , then G is a *true* sentence, and in a complete logic, all *true* sentences are provable, so B should be *true* . We can conclude from this simpler construction that if a logic is expressive enough to define B , then the logic is either inconsistent or incomplete.

Even this simpler construction includes one more definition than necessary. We could define

B = *true* if B=*false* is a provable binary expression;
false otherwise

If we suppose *B* is *true*, then *B=false* is a *false* sentence, and in a consistent logic, no *false* sentence is provable, so *B* should be *false*. If we suppose *B* is *false*, then *B=false* is a *true* sentence, and in a complete logic, all *true* sentences are provable, so *B* should be *true*. If a logic is expressive enough to define *B*, then the logic is either inconsistent or incomplete.

Turing

Epimenides talked about truth; Gödel talked about provability; Turing talked about computability using the same sort of arguments [12]. For my examples, I will use the Pascal programming language, but the choice of language is irrelevant; any other programming language would do just as well. I'll start with a procedure named *twist* that is closely analogous to the Liar's Paradox.

```
procedure twist;
begin
    if (execution of twist terminates) then twist
end
```

I have not finished writing procedure *twist*; what remains is to replace the informal binary expression (execution of *twist* terminates) with either *true* or *false*, whichever one is correct. The problem in doing so is that the informal binary expression refers to itself in a self-contradictory manner: if the execution of procedure *twist* terminates, it should be replaced with *true*, creating a procedure whose execution does not terminate; if the execution of *twist* does not terminate, it should be replaced with *false*, creating a procedure whose execution does terminate. This is not a programming problem, not a computability problem, not a lack of expressiveness of Pascal. The problem is that the informal binary expression is an inconsistent specification. One might protest:

> Either execution of *twist* terminates, or it doesn't. If it terminates, use *true*; if it doesn't, use *false*. How can there possibly be an inconsistency?

But I hope the inconsistency is clear enough that no-one will protest.

As we did with the Liar's Paradox, let's present the same inconsistency as two declarations.

```
const halts = { true if execution of twist terminates, false otherwise };

procedure twist;
begin
    if halts then twist
end
```

The value of constant *halts* is missing. In place of the value there is a comment to specify what the value should be. If execution of procedure *twist* terminates, then the value should be *true*. If execution of procedure *twist* does not terminate, then the value should be *false*. So there is no problem in programming the value of *halts* after we determine whether the execution of *twist* terminates. If we suppose it does, then *halts=true*, and so we see that execution of *twist* does not terminate. If we suppose it does not, then *halts=false*, and so we see that execution of *twist* does terminate.

Procedure *twist* has been written in its entirety. Syntactically, it is a procedure; to determine that *halts* is being used correctly within *twist*, we need only the type of *halts*, not the value, and we have the type. Semantically, it is a procedure; to determine the meaning of *twist* we need only the specification of *halts*, not its implementation, and we have the specification. (That important programming principle enables a programmer to use pieces of programs written by other people, knowing only their specifications, not their implementations. It also enables a

programmer to change the implementation of part of a program, but still satisfying the specification, without knowing where and why the part is being used.) So there is nothing wrong with the definition of *twist*. The problem is that we cannot write the value of *halts* to satisfy its specification. This is not a programming problem, not a computability problem, not a lack of expressiveness of Pascal. The problem is that the specification of *halts* is inconsistent. One might protest:

> Either execution of *twist* terminates, or it doesn't. If it terminates, use *true*; if it doesn't, use *false*. How can there possibly be an inconsistency?

The inconsistency cannot be seen by looking only at *halts* or only at *twist*. Each refers to the other, and together they are inconsistent.

Let me complicate this inconsistency by adding a parameter so *halts* can say whether execution of any parameterless Pascal procedure terminates, not just *twist*. To pass procedures as data, we need to encode them in some way, and the easiest encoding is as a text. (Whenever programs are presented as input data to a compiler or interpreter, they are presented as texts.) We could pass the whole procedure as text, but it is simpler to pass just the procedure name as text, and to assume there is a dictionary of function and procedure definitions that is accessible to *halts*, so that the call *halts* ('*twist*') allows *halts* to look up '*twist*' and '*halts*' in the dictionary, and retrieve their texts for analysis. For later reference, call this version of *halts* and *twist* the "intermediate" version:

function *halts* (*p*: **string**): **boolean**;
{ return *true* if *p* represents a parameterless Pascal procedure whose execution terminates; }
{ return *false* otherwise }

procedure *twist*;
begin
    if *halts* ('*twist*') **then** *twist*
end

To determine that *twist* is syntactically a Pascal procedure, we need only the header for *halts*, not the body, and we have the header. To determine the semantics of *twist*, we need only the specification of *halts*, not its implementation, and we have the specification.

As before, we cannot write the body of *halts* to satisfy the specification. No matter how carefully worded it is, no matter how clear it sounds, the specification conceals an inconsistency. The inconsistency is revealed by applying *halts* to '*twist*'. If *halts* ('*twist*') = *true*, then execution of *twist* is nonterminating, so *halts* ('*twist*') should be *false*. If *halts* ('*twist*') = *false*, then execution of *twist* is terminating, so *halts* ('*twist*') should be *true*. This is still not a programming problem, not a computability problem, not a lack of expressiveness of Pascal. It is still the same inconsistency that was present in the unparameterized, unencoded version, and the same inconsistency that was present in the *twist* procedure. One might protest:

> Either execution of a procedure represented by *p* terminates, or it doesn't. If it terminates, *halts* (*p*) should return *true*; if it doesn't, *halts* (*p*) should return *false*. How can there possibly be an inconsistency?

Now the protest starts to sound more plausible because the parameter allows us to show a large number of examples which are not problematic. For example, if we define *stop* and *go* as

> **procedure** *stop*; **begin end**
> **procedure** *go*; **begin** *go* **end**

then

> *halts* ('*stop*') = *true*
> *halts* ('*go*') = *false*

These nonproblematic examples may fool us into believing that the specification of *halts*

makes sense. But they are irrelevant. Procedure *twist* shows us the inconsistency.

There is one last complication: a second parameter so *halts* can say whether execution of any Pascal procedure with an input parameter terminates.

```
function halts (p, i: string): boolean;
{ return true if p represents a Pascal procedure with one text input parameter }
{ whose execution terminates when given input i ; return false otherwise }

procedure twist (s: string);
begin
    if halts (s, s) then twist (s)
end
```

This is now a modern version of Turing's Halting Problem. Turing's argument is as follows.

> Assume that *halts* is computable, and that it has been programmed according to its specification. Does execution of *twist* ('*twist*') terminate? If it terminates, then *halts* ('*twist*', '*twist*') returns *true* , and so we see from the body of *twist* that execution of *twist* ('*twist*') does not terminate. If it does not terminate, then *halts* ('*twist*', '*twist*') returns *false* , and so we see from the body of *twist* that execution of *twist* ('*twist*') terminates. This is inconsistent. Therefore function *halts* cannot have been programmed according to its specification; *halts* is incomputable.

The two parameters (*p*, *i*) make a two-dimensional space, and point ('*twist*', '*twist*') is on its diagonal, which is why the argument is sometimes called a "diagonal argument". But any text would do equally well as a value for the second parameter, and the second parameter adds nothing to Turing's argument.

The surprise, and a main point of this paper, is that the computability assumption is unnecessary to the argument. Without assuming that *halts* is computable, I ask what the specification of *halts* says the result of *halts* ('*twist*', '*twist*') should be. If the specification says the result should be *true* , then the semantics of *twist* ('*twist*') is nontermination, so *halts* ('*twist*', '*twist*') should be *false* . If the specification says the result should be *false* , then the semantics of *twist* ('*twist*') is termination, so *halts* ('*twist*', '*twist*') should be *true* . This is inconsistent. Therefore *halts* cannot be programmed according to its specification. But the problem is not incomputability; it is inconsistency of specification [10][11]. It is the same inconsistency that was present in all previous versions, before I added the complications of parameters and encodings. It is just the Liar's Paradox in fancy clothing. In fact, Turing's argument could have been applied to the simplest version of *twist* with equal (in)validity.

```
procedure twist;
begin
    if (execution of twist terminates) then twist
end
```

> Assume that the expression (execution of *twist* terminates) is computable, and that it has been programmed according to its specification. Does execution of *twist* terminate? If it terminates, then (execution of *twist* terminates) is *true* , and so we see from the body of *twist* that its execution does not terminate. If it does not terminate, then (execution of *twist* terminates) is *false* , and so we see from the body of *twist* that its execution terminates. This is inconsistent. Therefore the expression (execution of *twist* terminates) cannot have been programmed according to its specification; it is incomputable.

But there are only two possibilities for programming (execution of *twist* terminates) ; they are *true* and *false* . Calling this choice "incomputable" says that one of them is correct but no computer program can determine which one. In fact, neither of them is correct, and that is called "an inconsistent specification".

Turing's argument can be applied to any property of program execution. For example,

procedure *twist1*;
begin
    if (execution of *twist1* prints 'A') **then** *print* ('B') **else** *print* ('A')
end

Termination of execution of *twist1* is not in question: when (execution of *twist1* prints 'A') is replaced with either *true* or *false* , whichever is appropriate, execution of *twist1* terminates. The question is whether 'A' or 'B' is printed. Turing's argument says that the property "prints 'A' " is incomputable, and so is every property of program execution (except for the trivial "always *true* " and "always *false* " properties) [9]. But the problem is not incomputability; the problem is inconsistency of specification.

Turing's argument can even be applied to a completely meaningless property of program execution. For example, "calumation" is a meaningless word. Suppose *Calumate* is a procedure whose execution has the calumation property, and *DoNotCalumate* is a procedure whose execution does not have the calumation property.

procedure *twist2*;
begin
    if (execution of *twist2* calumates) **then** *DoNotCalumate* **else** *Calumate*
end

Turing's argument says that the meaningless property calumation is incomputable.

Underdetermination

The Liar's Paradox, the Gödel sentence, and Halting Problem are all examples of inconsistency, which is also known as overdetermination. Here, "determination" means determining a solution. If there is no solution, we have overdetermination; if there is more than one solution, we have underdetermination. An example is the sentence

    This sentence is true.

Whereas the Liar's Paradox can be neither true nor false, the sentence just written can be either true or false. Giving the sentence the name *U* for underdetermined, it becomes the formula

    U = (*U*=*true*)

As an equation in unknown *U* , it has two solutions: both *true* and *false* . As a specification of *U* , it is consistent, but does not determine *U* .

Here is an example of the underdetermination of Gödel's provability specification.

B (*t*) = *true* if text *t* represents a provable binary expression;
false otherwise
H = " *B* (*H*) = *true* "

That is the same specification of *B* as before. Now we ask: What is the value of *B* (*H*) ? If we suppose *B* (*H*) = *true* , then *H* represents the sentence *true*=*true* , which is provable, so *B* (*H*) should be *true* , as supposed. If we suppose *B* (*H*) = *false* , then *H* represents the sentence *false*=*true* , which is not provable, so *B* (*H*) should be *false* , as supposed. The

specification of *B* is both overdetermined (for *G*) and underdetermined (for *H*).

Here is an example of the underdetermination of Turing's halting specification.

```
function halts (p, i: string): boolean;
{ return true if p represents a Pascal procedure with one text input parameter }
{ whose execution terminates when given input i; return false otherwise }

procedure straight (s: string);
begin
        if not halts (s, s) then straight (s)
end
```

That is the same *halts* specification as before; it says that the *halts* function will tell us whether the execution of a procedure terminates. What does it say about *straight*? If we suppose that *halts* ('*straight*', '*straight*') = *true*, we see from the body of *straight* that its execution terminates, as supposed, so that is a solution. If we suppose that *halts* ('*straight*', '*straight*') = *false*, we see from the body of *straight* that its execution does not terminate, as supposed, so that is also a solution. That is another inadequacy of the *halts* specification. The specification sounds just right: neither overdetermined nor underdetermined. But we are forced by the examples to admit that the specification is not as it sounds. In at least one instance (*twist*), the *halts* specification is overdetermined, and in at least one instance (*straight*), the *halts* specification is underdetermined.

Objective and Subjective Specifications

I now want to consider specifications of behavior, or activity, in general. I include human behavior, computer behavior, and other behavior. To keep the examples simple, I will use specifications that say what the output, or final state, of the behavior should be. And I will use specifications that relate input, or initial state, to output, or final state. The conclusions apply also to specifications that say what the interactions during the behavior should be, but my examples will not be that complicated.

A specification may have the form of a question, for example "What is two plus two?". Or it may have the form of a command, for example "Tell me what is two plus two.". The question and the command are equivalent because they invoke the same behavior. A specification may describe the desired behavior, for example, "saying what is two plus two".

A specification is <u>objective</u> if the specified behavior does not vary depending on the agent that performs it. For examples:

(0) Given a natural number, what is its square?
(1) What is the number of words in this question?
(2) What is the name of the first Turing Award winner?

For all three questions, the correct answer does not depend on who or what is answering.

A specification is <u>subjective</u> if the specified behavior varies depending on the agent that performs it. For examples:

(3) What is your name?
(4) What is your IP address?

The correct answer to question (3) depends on whom you ask. In this paper, "subjective" does not mean that the answer is a matter of disagreement, debate, doubt, or dishonesty. If we ask Alice what her name is, the answer "Alice" is correct, and all other answers are wrong. If we ask Bob the same question, the correct answer is different. Question (4) is similar to question (3), but applies to a computer rather than a human.

Subjectively and Objectively Inconsistent

Now consider this example:

(5) Lift Bob.

I am not interested in the variety of lifting techniques; I am interested only in the specified result: the agent lifts Bob. If we ask Hercules, who is very strong, to lift Bob, he can do so without difficulty. If we ask Alice, who is much smaller than Bob, she is not strong enough. The result is different, depending on who is trying to lift Bob. So it may seem that (5) is subjective. But the definition of subjective specification says "the specified behavior varies depending on the agent". When we ask Alice to lift Bob, we are asking for the same behavior (lifting Bob) as when we ask Hercules. So it may seem that (5) is objective. But suppose we ask Bob to lift Bob. He cannot do so, but not due to lack of strength. He cannot do so because the specification does not make sense when we ask Bob to lift himself. The specification makes sense for some agent (anyone other than Bob), and makes no sense for some agent (Bob). For that reason, (5) is subjective. If we restrict the set of agents to exclude Bob, then (5) is objective.

(6) Can Carol correctly answer "no" to this question?

Let's ask Carol. If she says "yes", she's saying that "no" is the correct answer for her, so "yes" is incorrect. If she says "no", she's saying that she cannot correctly answer "no", which is her answer. We are assuming for this and all subsequent questions that the only acceptable answers are "yes" and "no", and in this case, both answers are incorrect. Carol cannot answer the question correctly. Now let's ask Dave. He says "no", and he is correct because Carol cannot correctly answer "no". So (6) is subjective because it is a consistent, satisfiable specification for some agent (anyone other than Carol), and an inconsistent, unsatisfiable specification for some agent (Carol).

(7) Can any man correctly answer "no" to this question?

Let's ask Ed, who is a man. Suppose Ed says "no". Ed is saying that no man can correctly answer "no", and Ed, a man, is answering "no", so Ed is saying that his answer is incorrect. Suppose Ed says "yes". Ed is saying that some man, let's call him Frank, can correctly answer "no". But if Frank answers "no", he is saying that his own answer is incorrect. So Frank cannot say "no" correctly. So Ed's "yes" answer is incorrect. And the same goes for every man. But Gloria, who is not a man, can correctly say "no". Specification (7) is subjective because it is a consistent, satisfiable specification for some agent (anyone who is not a man), and an inconsistent, unsatisfiable specification for some agent (any man).

(8) Can anyone correctly answer "no" to this question?

If we ask Harry and he says "no", he is saying that his answer is incorrect. If he says "yes", he is saying that someone, let's say Irene, can correctly answer "no". But if Irene answers "no", she is saying that her answer is incorrect. So Harry can neither say "no" nor "yes" correctly. And the same goes for anyone else we ask. The correct answer to the question is therefore "no", but no-one can correctly say so (oops, I just did). I meant: no-one who is a possible agent can say so. I exclude myself from the set of possible agents just so that I can tell you that no possible agent can correctly answer "no". Specification (8) is objectively inconsistent.

Specifications (6), (7), and (8) are examples of twisted self-reference. The self-reference occurs when the specification talks about the agent who will perform the specification. The twist, in these examples, is the word "no". If we replace "no" with "yes" in these three specifications, then everyone can correctly answer "yes" to all of them, making them objectively consistent.

Church-Turing Thesis

If a specification can be computed by any one of:
- a Turing Machine (a kind of computer) [12]
- the lambda-calculus (a mathematical formalism) [3]
- general recursive functions (another mathematical formalism) [4][7]

then it can be computed by all of them; they all have the same computing power. The Church-Turing Thesis [13] says that each of these formalisms compute all that is computable. In a more modern version, the Church-Turing Thesis says that if a specification can be computed, then it can be computed by a program in any programming language. All programming languages provide the same computing power; each is equivalent to a Turing Machine.

Church and Turing were thinking of specifications of mathematical functions, like (0). It seems reasonable to me that the Church-Turing Thesis can be extended to all objective specifications. But its extension to subjective specifications comes up against a problem.

Reconsider subjective specification (7), but replace "man" with "Pascal program". (We define "Pascal program" in such a way that the question whether p is a Pascal program is decidable. And likewise for any other programming language.)

(9) Can any Pascal program correctly answer "no" to this question?

It's easy to write a Pascal program that prints "no". If that is the answer to (9), it is saying that there isn't a Pascal program that correctly answers "no" to the question, so in particular, the Pascal program that prints "no" doesn't give the correct answer. It's just as easy to write a Pascal program that prints "yes". If that program is the answer to (9), it says that "no" is the correct answer, so the Pascal program that prints "yes" doesn't give the correct answer either. In fact, the correct answer to (9) is "no", but no Pascal program can correctly say so. We can write a program in Python (which is another programming language) that prints "no" in answer to (9), and that answer is correct. No matter whether the agents are people or programs, the result is the same: one agent can satisfy the specification, but another can't.

The Church-Turing Thesis, in the version stated earlier, does not apply to subjective specifications. Specification (9) can be computed by a Python program, but not by a Pascal program.

A consequence of the Church-Turing Thesis is that any program in any programming language can be translated to a program in any other programming language. We'll look at program translation in a moment, but first, here is a non-programming example to illustrate the translation problem.

(10) Is this question in French?

The correct answer is "no". The question is easily translated into French.

(11) Cette question est-elle en français?

The correct answer is now "oui". Before translation, when the question is put to someone who understands the language the question is in, it invokes one behavior: saying "no". After an accurate translation, when the question is put to someone who understands the language the question is now in, it invokes a different behavior: saying "oui". Specifications (10) and (11) refer to a language, and changing the language of the question affects the answer.

Similarly, when we write a program to compute a subjective specification, then translate it to another language, it may invoke different behavior. This can occur when the specification refers to a programming language. First, here's an objective specification that refers to a programming language.

(12) Is text p a Pascal program?

Every compiler answers the question whether its input text is a program in the language that it compiles. Whether we write the program that computes (12) in Pascal or in Python, for the

same input *p* we should get the same answer. Specification (12) is objective, and the Church-Turing Thesis applies. Now replace the input with a self-reference.

(13) Is the program answering this question a Pascal program?

There are two ways to write a Python program to compute (13). The hard way is to give the program access to its own text, perform the lexical analysis and parsing and type checking and so on, just as a compiler would do, and then print the answer, which will be "no". The easy way is just to print "no" because that's the right answer. Now we translate our Python program to Pascal. If we programmed the hard way, the translated program accesses its own text, does the analysis, and prints the correct answer, which is "yes". If we programmed the easy way, the translated program prints "no", which is incorrect. Specification (13) is subjective, and the Church-Turing Thesis does not apply. Either the translation prints the correct answer by exhibiting different printing behavior, or the translation exhibits the same printing behavior and the answer is incorrect.

Yet another consequence of the Church-Turing Thesis is that in any programming language, you can write an interpreter for programs in any programming language. To pass a program as data to an interpreter, the program must be encoded, and the standard encoding is as text. Or we can pass just the program name as text, and provide a dictionary of program definitions, so that the name can be looked up and the full text retrieved.

Suppose we have an interpreter for Python programs written in Pascal. When the Python program that computes (13) the easy way, by just printing "no", is interpreted, the result is that "no" is printed. Is this answer correct? One might argue that the program answering the question is the Python program, and that the interpreter, written in Pascal, is just the execution mechanism. By this argument, the answer is correct. Or one might argue that the program being executed is the interpreter, and the Python program is just its text input. By this argument, the answer is incorrect. So the correct answer is unclear.

Suppose we interpret the Python program that computes (13) the hard way. Which text, the Python program or the interpreter in Pascal, gets analyzed? Again, the correct answer is unclear.

If *p* is a Python program, we can express its interpretation in Pascal as *interpret* ('*p*') . The language of *interpret* ('*p*') is Pascal, and '*p*' is a Pascal text (the Python text would be "*p*"). Therefore I adopt the view that the program being executed is the interpreter, while acknowledging that the other answer also has merit. This makes interpretation the same as executing a translation. Interpretation is therefore similarly limited to programs that satisfy objective specifications. Given a program that computes a subjective specification, its interpretation may produce behavior that differs from execution of the given program.

The same choice, whether to preserve the behavior or to preserve the specification, can occur without translation or interpretation, simply by renaming. For example,

(14) Given a text *p* representing a program, determine whether a call to the determining program appears within *p* .

Let's name the determining program *DoYouCallMe* . Given program *p* , it searches within *p* for a calling occurrence of *DoYouCallMe* , reporting "yes" if one is found, and "no" if not. Now let's rename the determining program *DoYouCallMe2* . If we just change the name of the program without changing what it is searching for, this name change preserves behavior, but the program no longer satisfies the specification (14). If we change both the program name and what it is searching for, the program still satisfies the specification (14), but its behavior changes: given the same input, *DoYouCallMe* and *DoYouCallMe2* may give different answers. If a program has access to its own name, then changing its name automatically changes what it is searching for; the result still satisfies (14), but has different behavior.

For objective specifications, I accept the Church-Turing Thesis that we can translate/interpret a program from any language to any other language preserving both the specification and the behavior. For subjective specifications we may not be able to preserve both. As we saw for specification (13), we may be able to choose which one we preserve. As we saw for specification (9), it may not be possible to preserve the specification. When a program's behavior depends on the language the program is written in, it may not be possible to preserve the behavior. This last statement might be best explained by making the programming language explicit. If program p is written in Python, and on input x computes output y, write p (Python, x) = y. Translating p to Pascal, or writing an interpreter in Pascal, write p (Pascal, x) = z. Even though the input to p remains the same x, the output can change from y to z because the language has changed from Python to Pascal.

Halting Problem, Language-Based

Turing's Halting Problem is usually presented as applying to programs having an input. Without loss of generality and without changing the character of the problem, I consider halting for programs with no input; input to a program can always be replaced by a definition of a sequence of values, call it *input*, within the program.

(15) Given a text p representing a Pascal program that requires no input, report *true* if execution of p terminates, and *false* if execution of p does not terminate.

The input p represents a Pascal program. The agent that performs specification (15) must be a program, written in a programming language, running on a computer. I am excluding distributed computations so that I can identify the agent.

As we saw earlier in the "intermediate" version of *halts* and *twist*, we cannot write a Pascal program to satisfy specification (15). When programmed in Pascal, specification (15) is another twisted self-reference. The self-reference is indirect: *halts* applies to *twist*, and *twist* calls *halts*. The twist is supplied by *twist*. If *halts* reports that *twist*'s execution will terminate, then *twist*'s execution is a nonterminating loop. If *halts* reports that *twist*'s execution will not terminate, then *twist*'s execution terminates. Whatever *halts* reports about *twist*, it is wrong. Therefore specification (15) is inconsistent when we ask for a program written in Pascal to perform it [10][11].

Now let's ask for a program written in Python to perform (15). Can this Python program be written? Since Pascal programs cannot call Python programs, we cannot rule it out by a twisted self-reference. I present two possible answers to the question.

Answer O: Specification (15) is objective, like specification (12). But unlike (12), it is an inconsistent specification, no matter what language we use. If we could write a Python program to compute halting for all Pascal programs, we could translate it into Pascal (or interpret it by a Pascal program), and because (15) is objective, the translation (or interpretation) would also compute halting correctly for all Pascal programs. But there is no Pascal program to compute halting for all Pascal programs. So there is no program in any language to compute halting for all Pascal programs.

Answer S: Specification (15) is subjective. Like specification (13), (15) refers to programming language Pascal. When programmed in Pascal there is a twisted self-reference; when programmed in Python there is no self-reference. There is a Python program to compute halting for all Pascal programs. Because (15) is subjective, its translation to Pascal (or interpretation in Pascal) does not compute halting for all Pascal programs. Perhaps the Python program says correctly that *twist*'s execution terminates, and its translation to Pascal (or interpretation in Pascal), which we call *halts*, says incorrectly that *twist*'s execution does not terminate, and that is why *twist*'s execution terminates.

Answer O has been almost unanimously accepted by computer scientists, but its acceptance is premature because (15) has never been shown to be objective, and Answer S has never been ruled out. I favor Answer S for the weak reason that I cannot see any inconsistency in asking for a Python program to compute halting for all Pascal programs. (Writing the Python program would prove consistency. A logician says that's building a model; the logician's modeling language might be some version of set theory.)

Halting Problem, Location-Based

The preceding discussion of halting is language-based. Here is a similar discussion that is location-based. First a trivial example.

(16) Is this sentence written on page 1?

If (16) is written on page 1, the correct answer is "yes"; if it is written on page 2, the correct answer is "no". Although the answer depends on the location of the question, the answer does not depend on the agent answering, so it is an objective specification. We can create a subjective specification by creating a question that depends on the location of the agent answering.

There are some people in location A, and some other people in different location B. The question is:

(17) Can a person in location A correctly answer "no" to this question?

Anyone in location A who answers "no" to (17) contradicts themself. But Ingrid, who is standing in location B, can correctly answer "no" to (17) without self-contradiction. When Ingrid walks over to location A, she can no longer correctly answer "no" to (17). The question refers to Ingrid when Ingrid is at A; the question did not refer to Ingrid when Ingrid was at B. Even though she is the exact same person, with the same reasoning power, in either location, a correct answer in one location becomes incorrect in the other.

Suppose there are two identical disconnected computers C and D, and all programs are written in Pascal, and all programs can run on either computer. Both computers have enough memory so that memory limitation is not an issue. (Two computers are necessarily in different locations.)

(18) Given a text *p* representing a Pascal program that requires no input, loaded on computer C, report *true* if execution of *p* terminates, and *false* if execution of *p* does not terminate.

The agent that performs specification (18) must be a Pascal program running on either C or D. Once again, I exclude distributed computing so that I can identify the agent, and once again I assume there is a dictionary of function and procedure definitions on each computer.

First, let's ask for a Pascal program running on computer C to perform (18), and let's call it *halts* . If there is such a program, then we can write another program, let's call it *twist* , exactly as before, and we can load this program onto computer C. As before, *twist* calls *halts* to report on *twist* , and then *twist* does the opposite; so whatever *halts* reports, it is wrong. Specification (18) is inconsistent when we ask for a Pascal program running on computer C to perform it.

Now let's ask for a Pascal program running on computer D to perform (18). Can this program be written? Since programs on C cannot call programs on D (the computers are disconnected), we cannot rule it out by a twisted self-reference. As in the language-based case, we have the same two possible answers to the question: Answer O and Answer S.

Answer O: Specification (18) is objective. It is an inconsistent specification, no matter what computer we use. There is no program on any computer to compute halting for programs on computer C.

Answer S: Specification (18) is subjective. There is a Pascal program on computer D, and again let's call it *halts*, to compute halting for all Pascal programs on computer C. We can carry the *halts* program from D to C and run it there. But when we run it on C, it does not compute halting for all Pascal programs on C. This is quite counter-intuitive. When *halts* applies to *twist*, and *twist* calls *halts*, it matters whether the *halts* that applies (the first occurrence of *halts* in this sentence) is the same instance as the *halts* that is called (the second occurrence of *halts* in this sentence). In one case, there is a twisted self-reference, and in the other, there isn't, and that can affect the computation.

Normally, a program running on one computer will give the same answers to the same questions, with equal validity, as the exact same program running on another computer. This seems obvious, perhaps because it is true for objective specifications. But it is not always true for subjective specifications. The halting specification (18) is a twisted self-reference if the program answering it is on computer C, but not a self-reference if the program answering it is on computer D. So it seems probable that halting is subjective. Even if the program answering it is the exact same one on C and on D, a correct answer from the program running on D may be incorrect from the same program running on C. Furthermore, the same program running on C and D, with the same input, can give different answers to a question that refers to the location of the program.

Turing's Proof

Turing's proof that halting is incomputable appears on page 247 of [12]. The key paragraph is below. To help the modern reader, I have added the square bracketed words. Also, Turing used the word “machine” for the combination of hardware and software, and he used the words “universal machine” for the combination of interpreter program and computer. The first sentence is the assumption that halting is computable. The last sentence concludes that there was a self-contradiction (inconsistency), and therefore halting cannot be computed.

> Let us suppose that there is a such a process; that is to say, that we can invent a machine D [diagonal] which, when supplied with the S.D [standard description] of any computing machine M will test this S.D and if M is circular [nonterminating] will mark the S.D with the symbol "u" [unsatisfactory] and if it is circle-free [terminating] will mark it with "s" [satisfactory]. By combining the machines D and U [universal machine, or interpreter] we could construct a machine H [halting program] to compute the sequence beta' [a sequence that differs from the diagonal with U]. ... Now let K be the D.N [description number, or code] of H. What does H do in the Kth section of its motion? [What happens when H works on the representation of H?] It must test whether K is satisfactory, giving a verdict "s" or "u". Since K is the D.N of H and since H is circle- free, the verdict cannot be "u". On the other hand, the verdict cannot be "s". For if it were, then in the Kth section of its motion H would be bound to compute the first R(K-1)+1 = R(K) figures [R(n) is the number of terminating programs among the first n programs] of the sequence computed by the machine with K as its D.N and to write down the R(K)th as a figure of the sequence computed by H. The computation of the first R(K)-1 figures would be carried out all right, but the instructions for calculating the R(K)th would amount to "calculate the first R(K) figures computed by H and write down the R(K)th". This R(K)th figure would never be found. I.e., H is circular, contrary both to what we have found in the last paragraph and to the verdict "s". Thus both verdicts are impossible and we conclude that there can be no machine D.

Turing's proof does not appear to refer to a programming language, but implicitly it does. It talks about the standard description of a computing machine, which is a number that encodes a program. And Turing Machine programs can be numbered because they are in a language, the Turing Machine language, that has syntactic rules that enable us to enumerate programs. And

then a diagonal program D is assumed to be in that same enumeration, so it's in the same language, and the halting program H is constructed from D, so it's also in the same language. The proof fails to recognize the language dependence. It also fails to recognize location dependence by assuming there's only one computer. Turing's proof proves that there cannot be a program in the Turing Machine programming language, running on a Turing Machine, that determines halting for all programs in that same language running on that same machine. The possibility of computing halting for all programs in a set (language or location) by using a program outside the set (different language or different uncallable location) was not considered.

Other Proofs

There are several proofs that purport to prove that halting is incomputable [8]. The differences among them are superficial; at their core, they are all twisted self-references. The proof by Robert Boyer and J Moore [2] is distinguished by their claim that it is completely formalized and verified using an automated prover, ACL. ACL is a constructive logic in which all recursions must be well-founded to ensure termination. They define the bounded halting program $B(p, n)$ saying whether execution of p terminates within n steps, but they cannot define the halting program (*nat* is the natural numbers)

$$H(p) = \exists n\text{: } nat\cdot\ B(p, n)$$

in ACL because they lack quantification over an infinite domain. In place of Turing Machine operations, they use LISP programs, which are defined by writing a bounded EVAL function to interpret LISP. When execution of p runs past n steps, EVAL (p, n) returns the result BTM . So "execution of p fails to halt" becomes

$$\forall n\text{: } nat\cdot\ \text{EVAL}(p, n) = \text{BTM}$$

which cannot be expressed in ACL due to the unbounded quantification, but which can be proven by induction for any choice of nonterminating p . The gap between a constructive prover and an essentially classical (nonconstructive) theorem is filled with convincing but informal reasoning, so the proof is sound but not fully formal.

In place of a numeric encoding of programs, they use a textual encoding, as does this paper. And they define function CIRC exactly the same as the definition of *twist* in this paper, but in LISP rather than Pascal.

```
CIRC (A)
   ( IF ( HALTS (QUOTE (CIRC A))
                ( LIST ( CONS (QUOTE A)
                              A ) )
                A )
        (LOOP)
        T )
```

The theorem they prove, paraphrased roughly, says: If a program named HALTS behaves like the halting function (returning T for programs that halt and F for those that don't), then HALTS applied to CIRC returns BTM . This is inconsistent, therefore there is no LISP function to compute halting for all LISP functions. The same conclusion applies to any programming language. But again, the possibility of computing halting for all programs in a set by using a program outside the set was not considered.

Here is another proof that appears in textbooks without mentioning a dependence on a programming language.

[start of proof] All programs are finite sequences of characters, although not all finite sequences of characters are programs. Execution of a program may read characters as input, may write characters as output, and either terminates or computes forever.

Let C be a finite character set, and let C^* be the set of all finite sequences of characters in C. Define the mathematical function H (not a program) called "the halting function" as follows.

H: $C^* \times C^* \rightarrow \{true, false\}$

$H(p, i)$ = *true* if p is a program with one text input and execution of $p(i)$ terminates;
false otherwise

If p is a program whose execution on input i terminates, then $H(p, i) = true$, whether or not the entire input i is read. If p is a program whose execution reads the entire input i and waits forever for more input, then $H(p, i) = false$. If p is not a program with one text input, then $H(p, i) = false$.

Is there a program *twist* with one text input having the following behavior? For all p in C^*,
- if $H(p, p) = false$ then execution of *twist* (p) terminates;
- if $H(p, p) = true$ then execution of *twist* (p) does not terminate.

If execution of program *twist* (*twist*) terminates, then according to the definition of H, H (*twist*, *twist*) = *true*, not *false*. And if execution of program *twist* (*twist*) does not terminate, then according to the definition of H, H (*twist*, *twist*) = *false*, not *true*. So there is no such *twist* program.

Assume (for contradiction) that H is computed by a program *halts*. Then we can write program *twist* as follows.

Execute *halts* (p, p) but don't output.
If the output from executing *halts* on p would be *false*, terminate execution.
If the output from executing *halts* (p, p) would be *true*, loop forever.

But there is no such *twist* program. Therefore there is no such *halts* program; H cannot be computed by a program. [end of proof]

There are three criticisms of this proof. The first is that it fails to distinguish between a program and a text encoding of the program. Gödel and Turing both understood the importance of that distinction, although they used numeric encodings because the text (character string) data type had not yet been invented when they did their work. To see the difference, consider the arithmetic expression 1+2 and the text "1+2". The former is equal to 3, but the latter is not equal to 3, nor is it equal to "3".

1+2 = 3
"1+2" ≠ "3"

In the formal methods community, we treat programs as mathematical expressions. For example,

x:= 2; y:= 3 = y:= 3; x:= 2

because execution of the program on the left has exactly the same effect as execution of the program on the right. But

"x:= 2; y:= 3" ≠ "y:= 3; x:= 2"

because they are different texts. Although the proof fails to distinguish, fortunately it does no harm; we just need to reword the proof. For example, "if p is a program" becomes "if p is a text representing (or encoding) a program". The proof achieves an economy of expression by not distinguishing between a program and the text representing the program.

The second criticism of the proof is the failure to recognize its dependence on a programming language. When H is defined using the phrase "if p is (a text representing) a program", we need to know the rules of program formation; in other words, we need to know the programming language. And since we apply H to *twist*, we are assuming that *twist* is in that same language. When we program *twist*, it calls *halts*, so we are assuming *halts* is callable

from *twist* . The conclusion should have been that *H* cannot be computed by a program in the language over which *H* is defined.

The last criticism of the proof is that it's unnecessarily complicated. *H* and *twist* and *halts* do not need input parameters; they could be defined for only the one input they are applied to in the proof.

Oracles, Translations, and Interpreters

The Halting Problem is this: there is a mathematical halting function that says, for each Pascal program, whether its execution terminates; but there is no Pascal program to implement this mathematical function. (Substitute Turing Machine language, or any other programming language, for Pascal.) The reason, according to standard theoretical computer science, is the limited power of computation, compared to the (unlimited?) power of mathematics. There is a field of research called hypercomputation that studies computation strengthened by magical powers. There are many journal articles and books on the subject. This field was begun by Turing in [13]; he strengthened the power of computation by adding an oracle to determine halting. It works as follows, except that I refer to the Pascal language instead of the Turing Machine language, and I am using text parameters instead of numeric parameters.

[start] Let $H(p, x)$ be the mathematical halting function. Parameter p is a text representing a Pascal procedure with one text parameter x .
- $H(p, x)$ = *true* if execution of the procedure represented by p on input x terminates
- $H(p, x)$ = *false* if execution of the procedure represented by p on input x does not terminate

Fortify Pascal with *oracle* defined such that *oracle* ('*H*', '*twistO*', *x*) = *H* ('*twistO*', *x*) , and simultaneously fortify *H* to apply to fortified Pascal procedure *twistO* , defined as

procedure *twistO* (*x*: **string**); **begin if** *oracle* ('*H*', '*twistO*', *x*) **then** *twistO* (*x*) **end** [end]

As we have seen many times, there is an inconsistency: the specifications of *H* and *oracle* together are inconsistent. If we blame the inconsistency on the specification of *H* , then there is no mathematical halting function, so we cannot conclude that "the halting function" is incomputable. So it is commonly agreed to blame the inconsistency on the specification of *oracle* , and conclude that *oracle* cannot be programmed in Pascal. It is further commonly concluded that the reason *oracle* cannot be programmed is that computation power is limited, and cannot compute functions defined using more powerful mathematics.

As we have done before, we can see the inconsistency more clearly if we get rid of the parameters.

[start] Let *H* be the mathematical binary value such that
- *H=true* if execution of Pascal procedure *twistO* terminates
- *H=false* if execution of Pascal procedure *twistO* does not terminate

Fortify Pascal with *oracle* defined as *oracle=H* , and simultaneously fortify *H* to apply to fortified Pascal procedure *twistO* , defined as

procedure *twistO*; **begin if** *oracle* **then** *twistO* **end** [end]

The definitions of *H* and *oracle* together are inconsistent. We can eliminate *oracle* , and fortify Pascal with the mathematical halting function:

[start] Let *H* be the mathematical binary value such that
- *H=true* if execution of fortified Pascal procedure *twistH* terminates
- *H=false* if execution of fortified Pascal procedure *twistH* does not terminate

Fortify Pascal with *H* .

procedure *twistH*; **begin if** *H* **then** *twistH* **end** [end]

And the inconsistency remains. The proof of inconsistency is similar to the Liar's Paradox; it does not use the supposed power advantage of mathematics over computation (whatever that might mean).

Let us now repeat the argument, but replacing the mathematical function with a Python function. With parameters,

[start]
def *haltsPy* (*p*, *x*):
"""return **True** if execution of the procedure represented by *p* on input *x*
terminates; return **False** otherwise"""

procedure *twistT* (*x*: **string**); **begin if** *translate* ('*haltsPy*', '*twistT*', *x*) **then** *twistT* (*x*) **end**

where *translate* ('*haltsPy*', '*twistT*', *x*) is the translation from Python to Pascal of *haltsPy* ('*twistT*', *x*) . [end]

Without parameters,

[start]
def *haltsPy*: """return **True** if execution of the Pascal procedure *TwistT* terminates;
return **False** otherwise"""

procedure *twistT*; **begin if** *translate* ('*haltsPy*') **then** *twistT* **end**

where *translate* ('*haltsPy*') is the translation from Python to Pascal of *haltsPy* ; **True** is translated to *true* , and **False** is translated to *false* . [end]

In this example, *haltsPy* plays the same role as *H* played previously, and *translate* plays the same role as *oracle* played. The proof of inconsistency is identical to previously. The specifications of *haltsPy* and *translate* together are inconsistent. But this time, inexplicably, the usual conclusion is opposite to previously: *haltsPy* is blamed, not *translate* . It is commonly concluded that we cannot program *haltsPy* because of the limited power of computation, but if we could, we could translate *haltsPy* preserving both the behavior (same results) and specification (it still tells us whether *twistT* halts). These conclusions are unwarranted because the power of computation did not enter the argument, and we have seen that translation sometimes does not preserve both behavior and specification.

Using interpretation instead of translation, with parameters,

procedure *twistI* (*x*: **string**); **begin if** *interpret* ('*haltsPy*', '*twistI*', *x*) **then** *twistI* (*x*) **end**

where *interpret* ('*haltsPy*', '*twistI*', *x*) is the Pascal interpretation of the Python function *haltsPy* ('*twistI*', *x*) .

Without parameters,

procedure *twistI*; **begin if** *interpret* **then** *twistI* **end**

where *interpret* is the Pascal interpretation of the Python function *haltsPy* .

The arguments, common conclusions, and my criticisms are the same for interpretation as for translation.

Conclusion

The Epimenides construction shows us that asking for a function whose result is *true* for all and only those texts representing true sentences in a sufficiently expressive language is both overdetermined (inconsistent) and underdetermined. The Gödel construction shows us that asking for a function whose result is *true* for all and only those texts representing provable sentences in a sufficiently expressive language is both overdetermined (inconsistent) and underdetermined. The Turing construction shows us that asking for a function, written in a programming language, whose result is *true* for all and only those texts representing procedures, written in that same language, whose execution terminates, is both overdetermined (inconsistent) and underdetermined.

A specification is objective if the specified behavior does not depend on the agent that performs it, and subjective if it does. The Church-Turing Thesis applies to objective specifications, not to subjective ones. If an objective specification can be implemented as a program in a programming language, it can translated to a program in any other programming language, preserving both the specification and the behavior. If subjective specifications are implemented as programs in a programming language, it is sometimes impossible to translate them to programs in another programming language, preserving both the specification and the behavior.

In 1936, when Turing wrote the paper [12] including his proof that halting cannot be computed for all programs, there were no computers. He imagined one computer, and described his design. There were no programming languages except the machine language for his machine. With one machine and one language, he concluded correctly that halting cannot be computed. He did not consider the possibility of more machines and more languages.

Let X and Y be two programming languages, or two computers. It is inconsistent to ask for an X-program to compute halting for all X-programs due to a twisted self-reference. Twisted self-reference is characteristic of subjective specifications. So it may be consistent and satisfiable to ask for a Y-program to compute halting for all X-programs. At least it has not yet been proven impossible.

References

[0] The title of this paper pays homage to the wonderful book by Douglas R. Hofstadter: *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, 1979
[1] The Jerusalem Bible, Reader's Edition, Titus, chapter 1 verse 12
[2] R.S.Boyer, J S.Moore: a Mechanical Proof of the Unsolvability of the Halting Problem, *J.ACM* v.31 n.3 p.441-458, 1984 July
[3] A.Church: *the Calculi of Lambda Conversion*, Princeton University Press, 1941
[4] S.B.Cooper: *Computability Theory*, Chapman&Hall, 2004
[5] Epimenides Paradox, Wikipedia, http://en.wikipedia.org/wiki/Epimenides_paradox
[6] K.Gödel: über Formal Unentscheidbare Sätze de Principia Mathematica und Verwandter Systeme I, *Monatshefte für Mathematik und Physik* v.38 p.173-198, Leipzig, 1931
[7] K.Gödel, reported by S.C.Kleene: *Introduction to Metamathematics*, North-Holland, 1951
[8] E.C.R.Hehner: the Halting Collection, hehner.ca/HC.pdf

[9] H.G.Rice: Classes of Recursively Enumerable Sets and their Decision Problems, *Transactions of the American Mathematical Society* v.74 p.358-366, 1953
[10] W.Stoddart: "the Halting Paradox", FACS FACTS: the Newsletter of the Formal Aspects of Computing Science Specialist Group, 2018 January
[11] W.Stoddart: Halting Misconceived, EuroForth 2017, http://www.complang.tuwien.ac.at/anton/euroforth/ef17/papers/stoddart.pdf
[12] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* s.2 v.42 p.230-265, 1936; correction s.2 v.43 p.544-546, 1937
[13] A.M.Turing: Systems of Logic based on Ordinals, p.8, Ph.D. thesis, Princeton University, 1939
[14] A.N.Whitehead, B.Russell: *Principia Mathematica*, Cambridge University Press, 1910

Acknowledgements

I thank Bill Stoddart for stimulating discussions, and an anonymous referee for causing me to improve my paper.

other papers on halting

Note added 2020-12-9

I translated
(10) Is this question in French?
to French as
(11) Cette question est-elle en français?
In (10), the words "this question" clearly refer to the question they are part of: question (10). It might be argued that in (11) the words "Cette question", being a translation of "this question", refer to the same question that "this question" refers to, which is question (10). If so, then the answer to (11) is "non", which is a translation of the answer to (10).

In any decent programming language, we can define a function recursively. (Even the simplest arithmetic operations, such as counting, addition, and multiplication, must be defined recursively.) In language M we can define function f such that the body of the definition calls f. In the body, f refers to the function that it is part of. When we translate from language M to language L, we again define f such that the body of the definition calls f. In the body, f refers to the L-function that it is part of, not to the M-function that we are translating. So, to be like program translation, I took the words "Cette question" to refer to the question they are part of, which is question (11). As a result, the translated question (11) invokes a different behavior: saying "oui", which is not the translation of the answer to (10).